The Case for Precision Sharing

Sailesh Krishnamurthy, Michael J. Franklin, Joseph M. Hellerstein, and Garrett Jacobson

IRB-TR-04-006

February, 2004

Intel Research Berkeley

The Case for Precision Sharing

Sailesh Krishnamurthy Michael J. Franklin Joseph M. Hellerstein Garrett Jacobson

Computer Science Division, Department of EECS
University of California, Berkeley
{sailesh,franklin,jmh,garrettj}@cs.berkeley.edu

Abstract

Sharing has emerged as a key idea of stream query processing in static (e.g., NiagaraCQ, Aurora, STREAM) and adaptive (e.g., CACQ, PSoup) dataflow systems.

Inherent in all these schemes is a tension between *sharing common work* and avoiding *unnecessary work*. Increased sharing has generally led to more unnecessary work.

We present our notion of *precision sharing* which aims to share aggressively *without* unnecessary work. We show that tuple lineage, from adaptive systems, has more general applicability. Specifically, we demonstrate how to use it in static dataflows to reach precision sharing. We also show how static ordering constraints can be used for precision sharing in adaptive systems. Finally our lineage schemes have space and time overheads that we study experimentally.

1 Introduction

Data stream management systems support continuous queries that are long-running. Since multiple queries are active at concurrently over common data streams, shared processing is very attractive.

Two approaches to shared stream processing have emerged. In systems like NiagaraCQ [6], Aurora [3] and STREAM [14], tuples flow through *static* dataflow networks. In contrast, the idea of *adaptive* query processing has led to systems like CACQ [12], TelegraphCQ [5] and "distributed eddies" [18] where tuples are variably routed through an adaptive network.

The motivation for sharing with streams is the same as stated by Sellis [16] for classical systems: "to limit the redundancy due to accessing the same data multiple times in different queries." In Figure 1, we illustrate this redundancy with an example involving two queries.

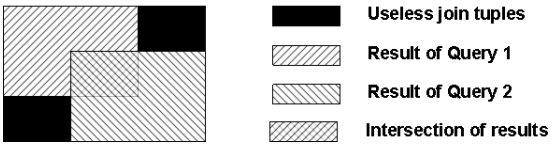


Figure 1: Sharing 2 queries: redundancy and waste

In the example, the result sets of the two queries overlap. Without sharing, the overlapping tuples have to be produced twice - a redundancy. In attempting to avoid redundancy, however, current shared schemes produce too much data. In the figure, a shared scheme such as NiagaraCQ would produce the tuples in the entire rectangle, including the "useless tuples" in the two darkly shaded regions. Thus, it would appear that sharing has to balance the inherent tensions of:

- **Repeated work** caused by applying an operation multiple times for a given tuple, or its copies.
- **Wasted work** caused by the production and removal of "useless tuples".

While existing systems have taken this tension for granted, the goal of our paper is to show that this tension is not, in fact, irreconcilable; to design and implement techniques that resolve the tension in static and adaptive dataflows; and to experimentally verify these techniques.

1.1 Precision Sharing

We define *precision sharing* as a way of characterizing any shared query processing scheme. We show that when sharing is precise, it is possible to avoid the overheads of repeated work *as well as* that of wasted work. Precision sharing applies to static as well as adaptive streaming systems, and is orthogonal to query optimization. It can also be used with traditional multiple-query optimization (MQO) schemes.

Static shared dataflows

We start by studying how the static shared plans described in NiagaraCQ are imprecise. Our technique to make such plans precise, uses the idea of tuple lineage from the adaptive query processing literature. While lineage has

**Proceedings of the 30th VLDB Conference,
Toronto, Canada, 2004**

been thought of as something useful in highly variable environments, our insight is that it is more generally applicable. Specifically, we show we can use tuple lineage to make static shared dataflows precise. We call our approach TULIP, or TUple LIneage in Plans.

Adaptive shared dataflows

Next we show how the CACQ shared adaptive dataflow system is also imprecise. Our strategy toward adaptive precision sharing is to borrow from the static world. We show how we can place constraints on how tuples are routed in an adaptive scheme to ensure that sharing is precise. Our approach is CAR, or Constrained Adaptive Routing.

We implemented both schemes, TULIP and CAR, in the TelegraphCQ system that we are building at Berkeley.

1.2 Contributions

Our contributions in this paper are to:

1. Argue that the tension between avoiding the overheads of repeated work and wasteful work in sharing is not irreconcilable, and define *precision sharing* to show how both overheads can be reduced in tandem.
2. Demonstrate the general utility of tuple lineage beyond adaptive query processing, and show how it can be used to achieve static precision sharing.
3. Show how to implement adaptive precision sharing with proper operator routing.
4. Validate our claims experimentally.

The rest of this paper is organized as follows. We briefly describe relevant work on shared stream processing in Section 2. Next, in Section 3, we define precision sharing and explain pitfalls in prior art. This is followed by a description of TULIP in Section 4 and a study of its performance in Section 5. We then present CAR in Section 6 followed by more experiments in Section 7. We end with a summary of our findings in Section 8.

2 Shared queries on streams

In this section we briefly describe the two major approaches to sharing: static query plans and adaptive dataflows. While sharing has also been studied in the multiple-query optimization (MQO) literature [16], there has been comparatively less work on shared processing of queries over data streams and the related topic of pipelined MQO [8, 17].

As has been well noted[12, 14], pipelined join operators are a natural fit for streaming query processors. For this reason we assume the exclusive use of symmetric join operators for the rest of this paper. This also simplifes the MQO problem by limiting the choice in join operators.

2.1 Static shared plans

The first approach we describe is the logical extension of traditional pipelined query plans to shared data stream processing. Here, a set of continuous queries is processed using a single static query plan that is a dataflow network of relational algebra operators. Figure 2 shows an example of a static dataflow that represents two shared queries.

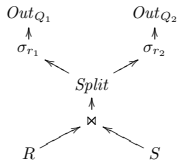


Figure 2: Static shared dataflow example

When new tuples arrive in the system they are driven through the network according to an operator scheduling policy. While different operators may be executed at different times, the *paths* taken by a tuple from a given stream to its various destinations are always the same[1]. Sharing is thus determined entirely by sub-expressions that are common to individual queries. This model has been adopted by NiagaraCQ, Aurora, and STREAM. NiagaraCQ [7, 6] describes ways to form *grouped* plans for multiple queries.

There are in general two approaches to MQO: (a) optimize each individual query and then look for sharing opportunities in the access plans, and (b) globally optimize all the queries to produce a shared access plan.

The first approach is easier to employ and is used in NiagaraCQ to group together plans for queries with similar structure. When new queries enter the system they are attached to an existing group whose *signature* it matches closely. A query that has many signatures is merged into multiple groups in the system.

2.2 Adaptive shared dataflows

The second approach we review is based on the idea of adaptive tuple routing and is a feature of systems like TelegraphCQ [5], CACQ [12] and PSoup [4]. In this approach too, a set of queries are decomposed into a dataflow of relational algebra operators. The major differences, however, are: (a) the dataflow is adaptive and can route tuples in a variety of different ways and (b) tuples are extended to carry their "lineage" consisting of "steering" and "completion" vectors, (c) the operators are aware of the completion vector of each input tuple - in other words two otherwise identical tuples with different completion vectors may be processed differently. We discuss adaptive dataflow technology in more detail in Section 6.

3 Precision Sharing

In this section we introduce and explain the importance of precision sharing, a way to characterize the overheads of shared query processing. We then show how current systems result in plans that are not precisely shared. We begin by defining precision sharing in terms of all operations performed on tuples in a shared dataflow.

[1]Work [10] on dynamism in static plans has generally been limited to one-time *late-binding* based on query parameters.

Precision sharing: A sharing scheme where for all inputs, the following properties *both* hold:

PS1 For each tuple processed, any given operation may be applied to it, or any copy of it, at most once.

PS2 For each tuple emitted by an operator, there must exist at least one query in whose results the tuple could possibly participate.

We clarify PS2 by considering the future of a tuple emitted by an operator. There are three possibilities: (1) The tuple is definitely part of the results of a query irrespective of any data from any source, (2) The tuple is part of the results of a query if it matches some condition(s) on tuple(s) from other source(s), and (3) The tuple is definitely *not* part of the results of a query irrespective of any data from any source. PS2 prohibits the production of tuples of the third kind. We call such tuples "zombies".

A plan that does not satisfy PS1 suffers from redundancy overheads. A plan that does not satisfy PS2 results in the wasteful *production* and subsequent *elimination* of zombies. We say that a given plan is *precisely shared* if it satisfies both the properties PS1 and PS2 for all inputs.

Conventional approaches in the MQO literature [16, 17, 8] have all assumed that reducing redundancy is paramount, without considering its side-effects. This definition of precision sharing lets us characterize the nature of such side-effects, and is essential to limiting unnecessary work for the query processor.

We now consider examples of imprecise sharing of join queries in the presence of selections on individual sources. We build on an example studied in NiagaraCQ [6, 7].

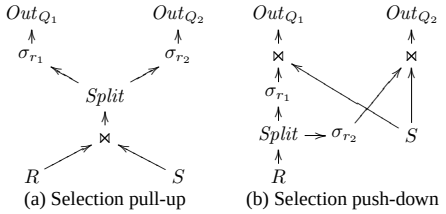
3.1 Imprecise sharing in action

Consider the following scenario involving two queries, Q_1 and Q_2, each of which join the streams R and S and apply a unique selection predicate on R.

- $Q_1 : \sigma_{r_1}(R) \bowtie S$
- $Q_2 : \sigma_{r_2}(R) \bowtie S$

NiagaraCQ suggests the two alternate plans for these queries. The plan in Figure 3(a) uses *selection pull-up* to share the RS join. In Figure 3(b) we see *selection push-down* where tuples in R are split according to the predicates first and then run in separate join groups. In actuality, NiagaraCQ combines the *Split* operator and its immediate downstream filters together, using an index for the filter predicates. We separate them for ease of exposition. Also, we use *Out* to represent a generic output operator that is equivalent to *TriggerAction* in NiagaraCQ.

Selection push-down violates PS1. For example, a tuple r_x from R that passes both predicates r_1 and r_2 will be processed in both join operators, producing identical join tuples. Also, every tuple from S will be built twice in each join operator (assuming symmetric hash joins). Multiple operations on such a tuple r_x, and all tuples in S are examples of PS1 violation. Each such example is an instance of redundancy in the plan. Note that in this example, PS2 is



(a) Selection pull-up (b) Selection push-down

Figure 3: Imprecise sharing of joins with selections

obeyed as each tuple from each join operator must satisfy at least one query.

Selection pull-up, on the other hand, violates PS2. For example, the output of the join operator can include an (r_x, s_x) tuple where r_x fails both predicates, r_1 and $r2$, satisfying neither query. The tuple (r_x, s_x) is an example of a zombie tuple, and shows how increased sharing can cause wasteful work. Note that this plan has only one join operator that produces the common sub-expression $R \bowtie S$ and has no redundancy. Since no operation is applied on any tuple more than once, this satisfies PS1.

We have seen how both pull-up and push-down violate at least one of the properties of precision sharing. A third alternative, however, was proposed in later work on NiagaraCQ [7]. This is a variant of pull-up called *filtered pull-up* which creates and then pushes down predicate disjunctions ahead of the join. In this example, the disjunctive predicate $(r_1 \vee r_2)$ is pushed down between the join and the scan on R. Such a plan is shown in Figure 4.

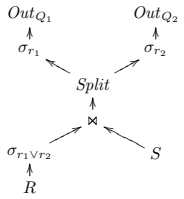


Figure 4: Precisely shared filtered pull-up

The disjunction is a "best-effort" predicate and is required only for performance and not for correctness, much like a bloom filter [2]. It is okay for the disjunction to let through"false positives".

Unlike pull-up, the filtered pull-up plan for this example satisfies PS2. This is because every R tuple r_x that reaches the join operator must have passed at least one of the r_1 and r_2 predicates. So every join tuple (r_x, s_x) must also satisfy at least one of the queries Q_1 and Q_2. Filtered pull-up also satisfies PS1 here for the same reasons as selection pull-up.

The filtered pull-up plan for this example satisfies both the properties PS1 and PS2. We now have an example of a sharing scheme that is precise. It is not surprising that the experimental and simulation results in NiagaraCQ [7] generally show this plan as the most efficient. It is reasonable to ask is if a filtered pull-up plan will always be precisely shared. It turns out that the answer is no, and we explain why in the next section.

3.2 Why filtered pull-up is not good enough

We now show why a filtered pull-up strategy is not precisely shared in general. We demonstrate this with an example where two queries, Q_3 and Q_4, join the streams R and S and apply unique selection predicates on *both* R and S. Notice that the only difference from the previous example is the selection predicate on S.

- $Q_3 : \sigma_{r_1}(R) \bowtie \sigma_{s_1}(S)$
- $Q_4 : \sigma_{r_2}(R) \bowtie \sigma_{s_2}(S)$

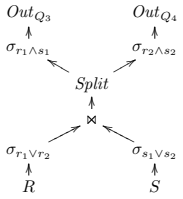


Figure 5: Imprecisely shared filtered pull-up

The filtered pull-up technique suggests that we pick the plan in Figure 5. The behavior of this query plan is shown in Figure 6. In the figure, R_1 and R_2 are respectively defined as $\sigma_{r_1}(R)$ and $\sigma_{r_2}(R)$. Similarly, S_1 and S_2 are respectively defined as $\sigma_{s_1}(S)$ and $\sigma_{s_2}(S)$.

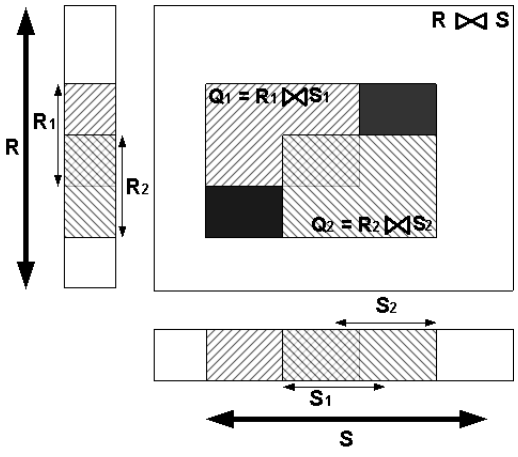


Figure 6: Filtered pull-up and zombies

Observe that the inputs to the join operator are the sets $R_1 \cup R_2$ and $S_1 \cup S_2$, and the join operator produces the set $(R_1 \cup R_2) \bowtie (S_1 \cup S_2)$. Notice that this is a superset of $(Q_1 \cup Q_2)$, our desired result. These extra tuples are zombies and are indicated in the figure as the two darkly shaded areas inside the smaller rectangle.

With two queries, it is easy to see the relationship between result set commonality and waste. As the intersection of Q_1 and Q_2 becomes larger, the wasted work gets less and vice versa. When more queries are added to the system, however, situations with high commonality and high waste are easily possible. In Figure 7 we show an illustration of such a scenario. The lightly shaded areas represent results of individual queries. The darkly shaded areas denote zombie tuples that are produced for no utility. In such cases, when there is both redundancy and waste, both the push-down and pull-up models are expensive.

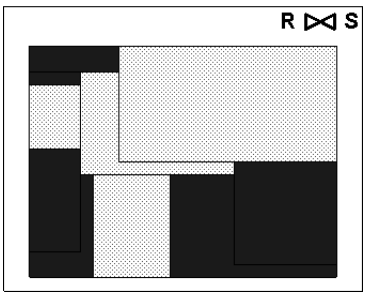


Figure 7: Zombies with many queries

The upshot of this example is that in spite of pushing down disjunctions, in the presence of *sharing*, a join can produce unnecessary *zombie* tuples that have to be eliminated later in the dataflow. With many queries this wasted work can increase significantly.

In the previous section we saw that the filtered pull-up strategy could share queries precisely. What we have seen here is that there are limits to the precision of this scheme. When the query was made slightly more complicated, with predicates on each source, property PS2 fails, and precision is lost.

In this example, the worst case overhead of lost precision is the maximal area of the region identified as the output of the shared join operator, i.e., $|R_1 \cup R_2| \times |S_1 \cup S_2|$. With two streams, the overhead is quadratic. As the number of streams increase, the overhead becomes more significant. In fact, it becomes *exponential* in the number of participating streams. We see more examples of this in the next section.

3.3 Disjunctions on intermediate results

We have shown how filtered pull-up can also cause the production of zombie tuples, violating property PS2. Now, we will show how zombies cause further inefficiencies when they participate in later join work producing even more zombies. Consider what happens when the queries in the example from Section 3.2 above also involve a third stream T.

- $Q_5 : \sigma_{r_1}(R) \bowtie \sigma_{s_1}(S) \bowtie \sigma_{t_1}(T)$
- $Q_6 : \sigma_{r_2}(R) \bowtie \sigma_{s_2}(S) \bowtie \sigma_{t_2}(T)$

A solution based on the pull-up strategy is to reuse the shared plan of Q_3 and Q_4 from Figure 5 and attach a join operator with T to each of Out_{Q_3} and Out_{Q_4}. That approach, however,could result in substantial duplicate join processing if there is significant overlap in the result sets of Q_3 and Q_4. This causes an appearance of a PS1 violation, which was not present in either of the pull-up schemes of the previous section. Given that the push-down plan already suffered from a PS2 violation, the resultant plan would be very inefficient.

The alternative is to discard the split from the plan shown in Figure 5 and use its input, complete with zom-

bies, in another shared join with T. This, however, exacerbates the zombie situation as the zombies that are input to the join cause even more zombies to be produced. These tuples will still ultimately be eliminated by the conjuncts evaluated at the top of the plan. Note that in this situation's worst case, the number of zombie tuples, is the product of the cardinality of the filtered sets of each source. With three sources, this overhead is cubic.

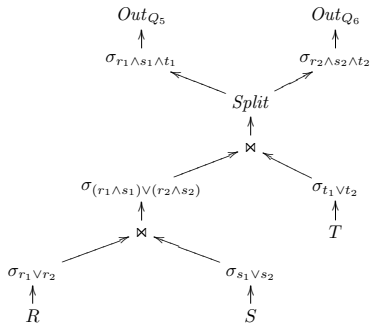


Figure 8: Eliminate zombies through disjunctions

This situation, i.e. the effects of zombies, can be ameliorated by pushing a *partial disjunction* down between the RS and ST join operators, assuming a left-deep strategy with an RST join order. In this case, this partial disjunctive predicate will be $(r_1 \wedge s_1) \vee (r_2 \wedge s_2)$. the plan is as shown in Figure 8.

Note that this plan still produces zombies after the RS join operator and still is in violation of PS2. In addition, a careful examination of this plan, reveals that the predicates r_1, r_2, s_1 and s_2 are each applied three times and t_1 and t_2 two times. This is a violation of PS1. With more streams being joined, the disjunction push-down scheme becomes increasingly complicated, suggesting that this approach is not very scalable.

Now suppose further that we are executing the queries Q_5 and Q_6 along with the queries Q_3 and Q_4. In keeping with our stated aim to share aggressively without generating zombies, we need to modify the plan in Figure 8 to produce the plan shown in Figure 9(a). Clearly the plan gets increasingly complicated with a lot of work being spent repeatedly re-evaluating predicates – the predicates on R and S are each potentially evaluated four times for a given tuple.

In addition to these violations of precision sharing, efficient execution of the Split operator is not easy. Recall that in actuality the Split operator is combined with all the predicates that are executed immediately after it. These predicates are built into a query index that the Split consults to route tuples. When the predicates involve more than one attribute, as is the case here, this index will have to be multi-dimensional.

In this section we showed how the standard techniques of shared query processing are not precise. In an attempt to efficiently reuse common work, they can end up producing useless data that can be exponential in the number of streams involved. Not only is the production of such useless tuples wasteful, the work done to eliminate them is an added waste.

We have shown how problems can occur in any kind of shared query processor, static or streaming. A particular manifestation of these problems is typically found in continuous queries that process *sliding windowed join* queries. Later in this paper we explain this scenario and show how our approach can handle it. Next, we focus on how to make sharing precise for static plans.

4 TULIP: Tuple Lineage in Plans

Based on the observations above, we propose TULIP: TUple LIneage in Plans, an approach that uses tuple lineage in static plans to achieve precision sharing.

4.1 A review of imprecise static sharing

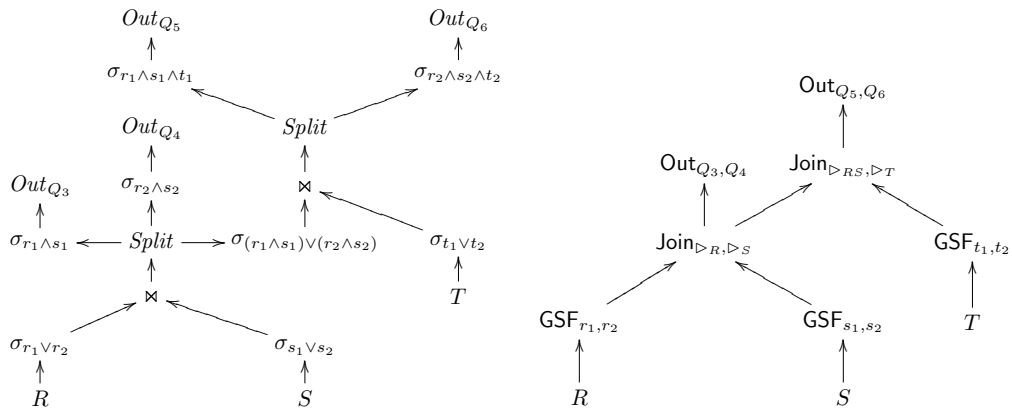
We saw in Section 3.3 why disjunctions on intermediate results can lead to complicated query plans with repeated predicate re-evaluation. Worse, these predicates evaluated on intermediate results are disjuncts of conjuncts – e.g. $(r_1 \wedge s_1) \vee (r_2 \wedge s_2)$ – and more expensive to evaluate than those that are disjuncts of simple predicates on base relations. This is especially so, when the number of queries is very large. We also saw how the filtered pull-up approaches can cause join operators to produce zombies, however early they can be eliminated. We summarize and then consider in turn each of these problems to guide us to our solution.

1. *PS1 violation in push-down:* When identical tuples reach different upper-level join groups the build and probe operations on the tuples are duplicated.
2. *PS1 violation in filtered pull-up:* The issue is that a predicate evaluation on a tuple, when successful, is likely to be repeated, potentially many times for complex queries.
3. *PS2 violation in pull-up:* In both the filtered pull-up and pull-up strategies, join operators can produce zombie tuples that have to be subsequently processed and eliminated.

With problem (1), the only time we can expect push-down to be competitive is when a very few upper-level join groups are activated for each base tuple. This observation was also made in NiagaraCQ [7]. The filtered pull-up strategies are the best way to reduce these overheads of repeated work and should be part of our solution.

Problem (2) arises because in static plans we throw away the results of earlier predicate evaluations. This makes sense in classical non-shared systems when predicates are generally conjuncts and the presence of a tuple above a filter is enough to deduce that the tuple passed every conjunct of the filter. Why not *memoize* the effect of each predicate evaluation and reuse it subsequently ? This is not dissimilar to index OR-ing strategies [13] for disjunctive predicates that are used in classical systems.

Problem (3) is again the result of discarding information on predicate evaluation. If, for each tuple, the information on each predicate evalution is memoized with the



(a) Many queries, many disjunctions

(b) Using TULIP for precision sharing

Figure 9: **To precisely share, or not to precisely share**

tuple, then a smart join operator can easily avoid producing zombie tuples.

With this problem analysis we are ready to describe our solution.

4.2 Tuple Lineage

We now consider the use of "tuple lineage" to accomplish memoization of predicate evaluation. Thus far, tuple lineage has been used profitably in adaptive query processing schemes. Our insight is that tuple lineage is more generally applicable, and is in fact useful in static dataflows.

As described in CACQ, all tuples that flow through the system carry lineage information that consists of: (1) a steering vector [1] that describes the operators in the dataflow that have been visited (`done`) and are to be visited (`ready`) and (2) a completion vector [12] that describes the queries in the system that are "dead" for this tuple, i.e., those that this tuple cannot satisfy. In CACQ, the distinction between these parts of lineage was blurred while in truth they have two distinct roles. The steering vector is entirely used as a tuple routing mechanism. Apart from the routing infrastructure, such as an Eddy operator, no other operator must use its contents. In contrast, the completion vector is a query sharing mechanism and should be entirely opaque to the routing fabric. In contrast, the other non-Eddy operators in the system are sensitive to the lineage of their input tuples.

The storage and manipulation costs of these vectors represent a major overhead in the tuple routing schemes. The completion vectors are particularly profligate in memory consumption – a bit per tuple per query results in space overhead that is linear in the product of the number of queries and currently active tuples. In contrast, when the queries in question share a lot of their operators, the steering vector size is much smaller.

4.3 The TULIP solution

Having defined the notion of tuple lineage, we are ready to present the TULIP solution. Our main tool is tuple lineage of which, we only need the "completion vector" part. For the rest of this paper, we refer to this portion as the "lineage vector".

The insight for the solution to Problem (2) is from Rete [9], a discrimination network for the many pattern/many match problem, the most time-consuming part of which is the match step. To avoid performing the same tests repeatedly, the Rete algorithm stores the result of the match with working memory as temporary state. The lineage vector that tags along each tuple keeps track of the queries that this tuple has already failed.

The same idea was also borrowed in CACQ with a grouped selection filter (GSFilter) operator that evaluates multiple similar predicates. The GSFilter maintains indexes on the conjunctive predicate clauses registered with it. When it receives a new tuple, it efficiently probes the index to identify all registered clauses that it fails. It then records all these failures in the tuple's lineage vector. If, at the end of processing the tuple, there still are any live queries for the tuple (i.e., queries that can still get satisfied) the tuple is sent to the output. The GSFilter implements the disjunction of the predicates and memoizes the results of each clause into the tuple's payload. All predicates are evaluated exactly *once*. Note that the GSFilter is doing *more* than what a simple disjunction would do. Apart from the disjunction it also sets up things so that the clauses of the disjunct need never be re-evaluated.

To eliminate the zombies of problem (3), we need a lineage-sensitive symmetric join operator with the following strategy: (a) ensure that tuples go through grouped filters prior to entering the join and (b) preserve the lineage vector of inner tuples when building into an index of the join. When an outer tuple probes the index and finds a matching inner tuple, we compute the union of the lineage vectors of the inner and outer. If this union consists of all queries that these operators are used by, then the match is

discarded.

To summarize, TULIP involves the following components:

1. Any appropriate MQO scheme that results in filtered pull-ups can be chosen to determine join orders.
2. The disjunctions that are pushed down should be replaced with GSFilter operators.
3. The symmetric join operators should be lineage sensitive and be able to recognize and remove zombies early.
4. Any operator that can be implemented in a lineage sensitive fashion to exploit sharing can be used in the plan.

We now put it all together for our driving example, the scenario that shares queries Q_3,Q_4, Q_5 and Q_6. The static query plan for the TULIP model is shown in Figure 9(b). We use three kinds of lineage sensitive operators. The GSF is a grouped selection filter, the Join is a zombie-killing symmetric join and the Out which is an output operator. The Out is similar to that used with the classic static plans except that it is a single operator that is capable of scheduling its input tuples to the appropriate delivery mechanism based on all live queries. First, PS1 is satisfied as this plan does not perform any operation on a given tuple more than once: all predicate evaluations are memoized in the lineage vectors of tuples and since the grouped filters push down disjunctions, no tuple is processed twice as part of a join operator. Next, PS2 is also satisfied as no join operators produce zombie tuples of any kind.

It is instructive to compare this plan with the equivalent traditional shared plan in Figure 9(a). Not only is the TULIP plan an example of precision sharing, it is easy to see how it can be scaled without much effort. In contrast, as we deal with more queries and streams, the filtered pull-up plan gets more and more complicated.

Our main insight in TULIP is that the use of lineage helps: (a) to memoize predicate evaluation and avoid repetitive computations, a lá Rete networks and (b) lineage sensitive operators to recognize and eliminate potential zombie tuples even before they are produced. These uses of tuple lineage ensure that TULIP does not respectively violate properties PS1 and PS2. In fact, TULIP guarantees precision sharing irrespective of optimizer decisions such as join order.

5 Performance of TULIP

In this section we study the performance of TULIP, our static precision sharing approach and compare the static schemes described in NiagaraCQ. In particular we consider the filtered pull-up and the selection pushdown schemes.

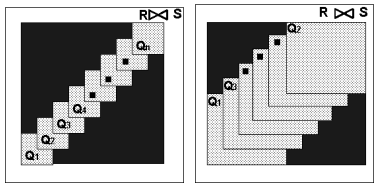
5.1 Experimental setup

Our experiments were performed on a 2.8 GHz Intel Pentium IV processor with 512 MB of main memory. We implemented TULIP in the TelegraphCQ [11, 5] system. Since we have no shared query optimizer, programmatically hook up static plans using the TelegraphCQ operators.

Specifically, for the static NiagaraCQ plans, we set up the system so that no lineage information is stored in intermediate tuples and that TelegraphCQ's lineage sensitive operators do not perform any unnecessary work. For instance, the disjunctions of filtered pull-up are realized with a GSFilter that does not set lineage. Similarly, a symmetric join operator is simulated with two SteMs [15] that again ignore lineage. The static plans shown in Section 3 have *Split* operators that are separate from the predicate filters that follow them, suggesting that each individual predicate is evaluated separately. However, in our experiments we follow the NiagaraCQ approach and use a Split operator that probes its input tuples into a predicate index implemented by a GSFilter. This lets Split send tuples only to those plan elements of queries that passed the probe. The top of each plan has one *Output* operator for each query.

In our TULIPimplementation, TelegraphCQ's intermediate tuples have lineage turned on. TULIP plans use GSFilters, implement lineage sensitive zombie-killing symmetric hash joins using SteMs [15], and use a lineage sensitive output operator.

In both implementations, the output operator makes a tuple available for delivery to a query by queueing it to the process managing the query's connection. The queue is in shared memory, access to which can be expensive. So, for all of these experiments we suppress output production. Even so, output processing is still not trivial. For latency computations, we make a system call to find the current time for each output tuple. This is still, however, cheaper than the actual system overheads of sending the same tuple multiple times through shared memory.



(a) Fewer overlaps (b) Greater overlaps

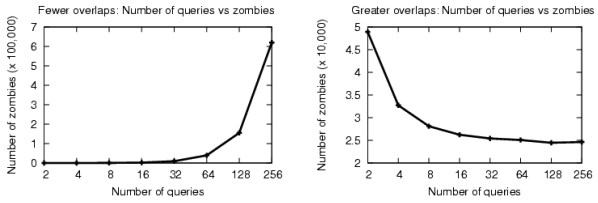
Figure 10: Experimental setup: Query result sets

```
select  R.a, R.b, S.a, S.b
from    R, S
where   R.a = S.a  AND
        R.b > const_0 AND R.b < const_1 AND
        S.b > const_2 AND S.b < const_3;
```

Figure 11: Experimental setup: query template

Our experiments all share a set of queries that are joins on streams R and S with individual predicates on each stream. The queries have identical structure and correspond to queries Q_3 and Q_4 from Section 3.2. The template of these queries is in Figure 11. We generate 256 queries for our experiments by supplying values for the constants in each of the queries in two setups. We show these visually in Figure 10. As before, shaded areas represent results of

queries and darkly shaded pieces are zombies that would be generated by selection pull-up. We used TULIP to log the number of zombies actually eliminated. This is shown for both cases in Figure 12.



(a) Fewer overlaps (b) Greater overlaps

Figure 12: Experimental setup: Zombies

In the first setup, shown in Figure 10(a), the result set of each query overlaps with few other sets. To be precise, each query's result set overlaps with that of two other queries.In this case, as queries are added in the system, more and more zombies are produced, as shown in Figure 12(a).

Conversely, in the second setup, shown in Figure 10(b), the result set of each query overlaps with many other sets. To achieve this, the first two queries are arranged so that they have almost no overlap (i.e., they are the two queries farthest apart). Subsequently, every query that is added overlaps with one or both of the first two queries. Since each such query contributes no extra zombies, the effect of adding queries is to steadily reduce the number of zombies produced, as shown in Figure 12(b).

In our experiments, we measure the average latency of each of the results of each query. Synthetic data is generated and piped into TelegraphCQ by an external process. Each tuple arriving at the system is timestamped on entry in the TelegraphCQ Wrapper ClearingHouse even before it is read by any scan operator. When a tuple arrives at an output operator, we examine its components and compute the difference between the current time and the time it originally entered the system. This represents the latency of the tuple, and the average latency is what we measured in our experiments.

We consider the 4 static approaches that we studied earlier: (a) selection pull-up (SPU), (b) filtered pull-up (FPU) (Figure 5), (c) selection push-down (SPD) and (d) TULIP. In our graphs, we only report the filtered pull-up case as its performance was always better, as expected, than selection pull-up. Plans for selection pull-up and push-down with predicates on only one source are shown in Figure 3 and the multiple predicate case is just a simple extension.

5.2 Performance results

For each setup, we plot in Figure 13 the average latency of result tuples for each approach against the number of queries being shared. Note that the number of queries is shown in a log_2 scale on the x-axis.

In both setups, the average latency for all plans is very small (under 25ms) for 2 queries and increases steadily as queries are added. In each approach, there is a certain number of queries at which there is a knee in the graph showing each scheme's scalability limits.

The following overheads affect average latencies:

- **PS1 violations:** Repeated work for the same tuples in intersecting result sets:
  (SPD) In the various separate join operators.
  (FPU) In output processing.
- **PS2 violations:** (FPU) Unnecessary work caused by the production of zombies in joins and removal afterward.
- **Other:** (TULIP) CPU instructions for lineage management. The state overhead was too small to burden virtual memory.

Setup 1 (Fewer overlaps):

As seen in Figure 13(a), for 32 or fewer queries the behavior of all three plans remains similar. Latencies increase steadily from 6ms to 17ms, while zombies produced by SPD increase increases from 14 to 9133.

At 64 queries, the latency for SPD jumps to 72 ms while that of FPU and TULIP stay at 30ms. For twice as many queries, the number of zombies increased four-fold to $\approx$ 39000. FPU's zombie overheads slow it materially and it scales no more for 128 and 256 queries. For these query sets its average latency is 430ms and 43 seconds.

Returning to FPU and TULIP, for more than 64 queries performance of both approaches start degrading. As queries are added, each new query causes more tuples that cannot be easily eliminated before joins. TULIP is, however, slightly more expensive than FPU and at 256 queries its latency is 147ms as opposed to FPU's 125ms.

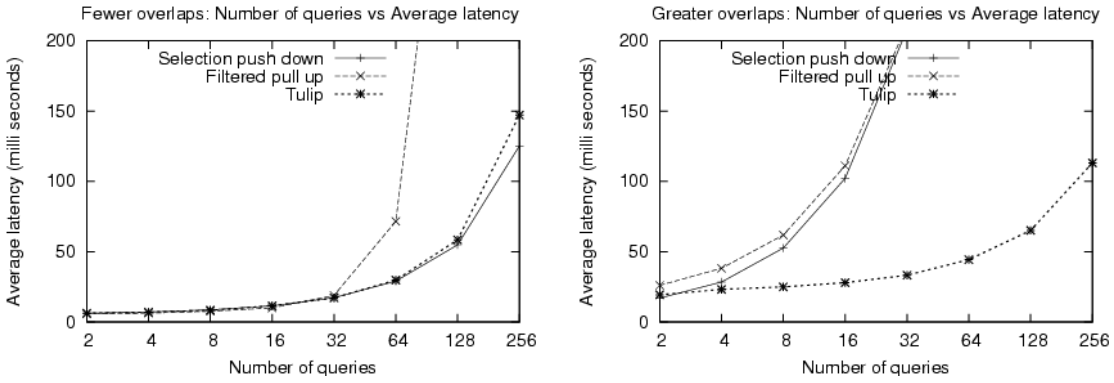
In general, sharing does not have much advantage when the results of the queries being shared have fewer overlaps. This is exactly what we observe in this case and the minimally shared SPD scheme does better overall. The repeated work overheads in SPD are slightly dominated by that of lineage management in TULIP. Both are comprehensively dwarfed by the zombie overheads of FPU.

Setup 2 (Greater overlaps):

As seen in Figure 13(b), all three plans behave similarly for 4 or fewer queries with latencies $\approx$ 25ms. For 2 queries, FPU is the outright winner as both queries have no overlap.

From 4 to 32 queries, the performance of FPU and SPD both degrade very fast. As queries are added, lots of tuples overlap causing repeated work. One instance of this is in output processing for which SPD and FPU behave similarly. These new tuples, however, also cause: (1) repeated join overheads in SPD and (2) overheads resulting from zombies in FPU. As zombies decrease from $\approx$ 49000 to plateau at $\approx$ 25000 the former overheads increase and the latter decrease. From 32 to 64 queries, both SPD and FPU perform the same. Beyond 64 queries, the join overheads of SPD become much worse, leading to SPD having a latency of 8.02 seconds for 128 queries as opposed to 1.7 seconds for FPU (these are not shown in the graph).

In contrast, the TULIP scheme performs very well, gracefully degrading in performance as the number of queries are added. At 256 queries, the latency of TULIP



(a) Fewer overlaps

(b) Greater overlaps

Figure 13: Static query plans: average query latencies

is 113ms. The FPU and SPD schemes have a comparable overhead of 111ms and 102ms for 16 queries. For the same latency, TULIP scales to 16 times, more than an order of magnitude, as many queries as traditional schemes.

Summary: The insights of our performance analysis are as follows:

1. The overheads of repeated work, *as well as* that of unnecessary work are significant.
2. Our two setups demonstrate two extreme cases, each favoring one of the two traditional approaches (FPU and SPD).
3. In each extreme case, the TULIP solution of precision sharing performs very well. While in the case of minimal sharing it is almost as competitive as the ideal FPU, in the face of high sharing it is more than an order of magnitude better than either traditional scheme.

In our experiments we worked with two extremes: one a setup where there is not much benefit to sharing, and the other where sharing can be very useful. The latter case brings out the tension between avoiding repeated and wasted work that current systems grapple with. Our experiments show that existing technology cannot deal well with this extreme, while our approach handles this easily. It is important, however, to get a feel of the costs of our approach - namely that of lineage management. The performance of TULIP in the former extreme case, i.e., without much overlap, shows that this extra overhead is not significant for the scenarios we studied. This suggests that TULIP is capable of giving very good benefits in the cases where current systems fail, while staying competitive otherwise.

6 Adaptive Precision Sharing

We begin this section by showing that adaptive shared schemes, while already including lineage, are also susceptible to violations in precision sharing. Just as we used ideas from the adaptive approach to make static sharing precise, it turns out that we can use techniques from the static world to remove the precision sharing violations from the adaptive approach.

6.1 Imprecisely shared adaptivity

In this section we show how sharing in an adaptive system like CACQ is also not precise. Here, we are concerned only with sharing and do not address for any of the considerable benefits that an adaptive system has.

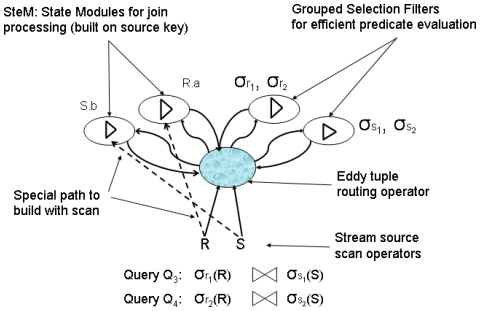


Figure 14: CACQ: Eddy, SteMs and Grouped Filter

In Figure 14 we show how CACQ will process the queries Q_3 and Q_4 from Section 3.2. Scan modules for R and S are scheduled to bring data in to the system from wrappers [11]. The tuples are fed into the eddy, which adaptively routes the tuples through its slave operators. There are two GSFilters, one each for all predicates over R and S, and two SteM operators. A SteM is a "state module" [15] that can be conceptualized as one half of a decoupled symmetric join operator. For e.g., a join operator $R \bowtie_a S$ over streams R and S may be decoupled into two SteMs $R.a$ and $S.a$.

In CACQ, a tuple is routed to candidate operators based on its signature - i.e., the set of base tuples that are its constituents. Operators amongst a set of candidates may be chosen in any order, with a routing policy governing this choice. A base tuple from R has a signature r and has to be *built* and *probed* into the R and S SteMs respectively. For correctness reasons, however, both SteMs cannot be used

as candidates for tuples with signature r as that will destroy the atomic "build then probe" property of pipelined joins. As described in Section 3.3.1 of CACQ [12], "a singleton tuple must be inserted into all its associated SteMs before it is routed to any of the other SteMs with which it needs to be joined". In fact, as shown in Figure 14, there is a special path for tuples to be built directly into their associated SteMs right after they have been scanned.

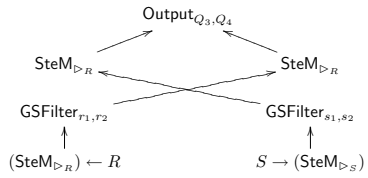


Figure 15: Effective tuple dataflow in CACQ

Thus, in this example, the adaptivity features of CACQ play no role, as there is only one join to be performed. In Figure 15 we show the dataflow of r and s tuples in CACQ for this example. A base tuple goes through Build, GSFilter, Probe and Output operators. Note that an r tuple gets respectively built and probed in the R and S SteM. Similarly an s tuple gets respectively built and probed in the S and R SteMs. For simplicity, we assume that the predicates in question are not expensive and so CACQ always orders the GSFilter before a Probe.

This scheme will never evaluate a predicate more than once as predicate evaluations are memoized into the tuple's lineage vectors. Although this does not violate property PS1 of precision sharing as far as predicate evaluation is concerned, we will see in Section 6.2 that the effects of output processing can cause PS1 violations.

Even so, the tuples that are built into SteMs are the original base tuples and do not contain any record of predicate evaluation. This, however, means that when producing the join tuples, there is no way to combine the lineage of the probe and build to eliminate zombies as described in Section 4. To see why this is so, recall from Section 4.3 our description of a lineage-sensitive symmetric join. To be able to able to eliminate matching tuples that are zombies, the operator needs to perform the union of the lineage vectors of the outer and inner tuples. In CACQ, however, the inner tuples are built before going through the GSFilter. As a result, CACQ cannot recognize any zombies and violates the PS2 property of precision sharing.

Explained in another way, this is a problem of the optimal placement of individual selection predicates in the presence of joins. There are two choices - pushing the selections down below the join, or pulling them above. In the presence of sharing the best way is another choice of combining selection pull-up with disjunction push down, which when accompanied by precision sharing is very efficient. In CACQ, however, SteMs are used to decouple join operators. So there are actually only two choices for locating the disjunction: either between the build and probe, or before the build. Put simply, CACQ made the former choice, and which is the wrong one in the light of precision sharing.

We now see that inspite of using tuple lineage as a mechanism for sharing, the CACQ scheme is not precisely shared for the same reasons as the pull-up strategies discussed earlier. This suggests, that lineage by itself is *not enough* for precision sharing. It is very important to route tuples in the right order for performance.

Our solution strategy is the inverse of what we did with static plans. A characteristic of static plans is that the route that tuples take through a dataflow impose an ordering on the operators of the network. We will show that we can apply similar ordering constraints on an adaptive dataflow to make its sharing precise.

6.2 Issues with the CACQ routing mechanism

We saw above that sharing in CACQ is not precise. Here we will show that the imprecision is a direct consequence of the routing mechanism in CACQ. As we explained above, the upshot of CACQ's "signature" routing mechanism is that "builds" are treated as an "out-of-band" operation. This is illustrated by the dotted lines indicating "special path" in Figure 14. If "builds" are out-of-band and excluded from the routing mechanism, filters cannot be executed before them, and so zombies cannot be recognized.

In addition to this problem, the routing mechanism has an undesirable effect on output processing. Without sharing, in the single query eddy scheme [1] the steering vector of a tuple indicates when its work is done and it can be output to a query. With sharing however, any intermediate tuple in the eddy might be ready for output processing. In CACQ this is done by comparing, in each major loop, an intermediate tuple's steering vector with the completion requirements of each query. Not only is this an expensive operation, especially in the presence of a large number of queries, there is a potential for a given tuple to be repeatedly processed as an output for multiple queries. We saw, however, in Section 5 that repeated processing of the same tuple in the outputs of multiple queries (PS1 violation) can drastically hurt performance. What we really need is a way to route tuples to output operators *only* when they are finally ready for them.

From this analysis it is clear that CACQ's main flaw is in its signature-based routing mechanism.

6.3 CAR: Constrained Adaptive Routing

Here, we propose as an alternative to CACQ, *Constrained Adaptive Routing*, or CAR. We will show that this scheme has all the adaptivity benefits of CACQ and still satisfy precision sharing.

The main difference between CAR and CACQ is in the routing mechanism. In CAR, we introduce the *operator precedence* routing mechanism. In this approach, we record precedence relationships between operators in an *precedence graph*. As with CACQ, this mechanism is used to generate a set of candidate operators to which tuples must be routed. In its simplest form, this is a graph with nodes that are sets of operators (called "candidates")

and edges that represent legal transitions from one node to the other. A transition is expressed with a signature - i.e., a tuple produced by an operator belonging to a node, may be routed. A given operator will generally exist in more than one node.

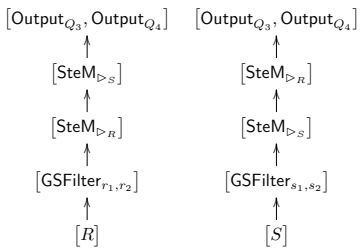


Figure 16: Operator precedence graph for CAR

In Figure 16 we show an operator precedence graph for the queries Q_3 and Q_4. There are 8 nodes in the graph and operators (such as the SteMs and Outputs) appear in more than one node. Clearly, with this scheme tuples are filtered and then built into SteMs, thus enabling the early recognition and elimination zombies. This prevents property PS2, and since PS1 is prevented for the same reasons as with CACQ, this plan is precisely shared.

Note that fixing predicate placement can hurt adaptivity. GSFilters ought to be processed before builds, but that is good only if the GSFilter is cheap. Irrespective of the mechanism to route tuples through operators, it would be nice to be able to adaptively move the location of selections. The issue is that the effects of placing a selection somewhere in the plan causes side-effects in terms of zombies elsewhere. It is not yet clear how to devise a routing policy that can use such *global* knowledge to adaptively choose the best location to place filters.

The main idea of the operator precedence routing mechanism is:

1. Subdivide the "namespace" of candidates that are keyed by a given signature.
2. Establish legal ordering relationships between operators, or sets of operators, that are keyed by the same signature.

This mechanism lets tuples be routed through sets of operators while enforcing any ordering demanded by correctness. The result is a scheme that allows precise sharing with all the benefits of adaptivity.

The main insight of this approach is our use of techniques from the static world. A purely adaptive approach makes routing decisions every step of the way. Constraints on the adaptivity makes it possible to ensure that predicate placement is appropriate for precision sharing.

7 Performance of CAR and CACQ

In this section we compare the performance of CACQ with CAR, the constrained adaptive routing technique we described above. Our experimental setup and methodology is identical to that described for static plans in Section 5.

For each of the two setups we report the average latencies of query results for each of CACQ and CAR in Figures 17(a) and 17(b). Note that as before, the number of queries is shown in a log_2 scale on the x-axis.

As in the static case, for both setups, the average latency of CACQ and CAR with 2 queries is small (5-30 ms) and increases steadily with query addition until scalability limits are reached.

The following overheads can affect latencies:

- **PS1 violations:** (CACQ) Repeated output processing of the same tuple in different queries.
- **PS2 violations:** (CACQ) Unnecessary work caused by the production and removal of zombies.
- **Other:** (CAR,CACQ) CPU instructions involving lineage management.

In this experiment, for CACQ the tuples produced by probes into SteMs are immediately ready for output. There are no more filtering steps and so there are, in fact, no PS1 violations causing output processing overheads.

In both setups, the performance of CAR comfortably outstrips that of CACQ. Just like TULIP, the performance of CAR gracefully degrades with the addition of new queries.

In the fewer overlaps case, with 2 queries there are actually no overlaps. In spite of this, the production of 14 zombies is enough to cause CACQ's latency to be 21 ms as opposed to 6ms for CAR. At 256 queries, the latency of CAR is $\approx$ 151 ms. An equivalent latency of CACQ supports only 18 queries. For this latency, CAR supports 14 times (an order of magnitude) more queries than CACQ.

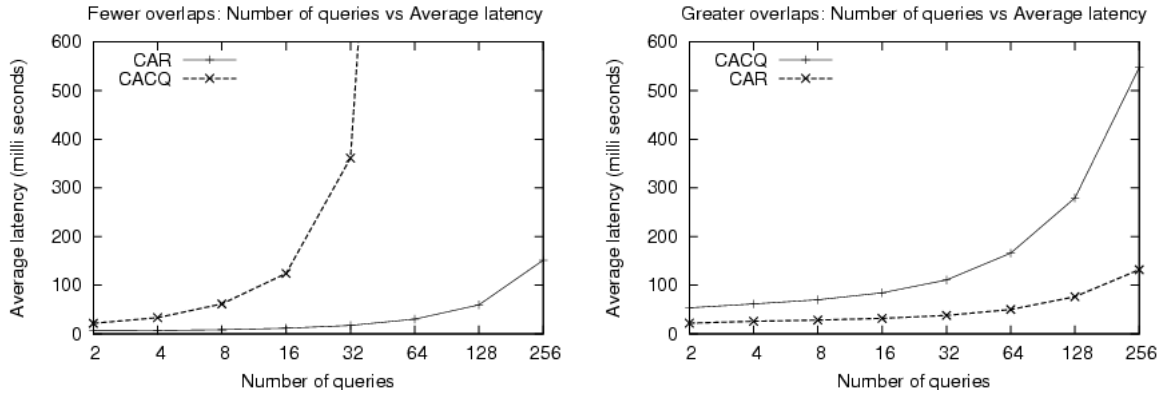
In the greater overlaps case, CACQ scales more gracefully than with fewer overlaps. Note that in this case, the relative overheads of zombies actually drop with more queries. The behavior of CACQ that we observe is really a *damped* version of CAR. With 256 queries CACQ has a latency of 550 ms as opposed to 131 ms of CAR. Note that CACQ can support a latency of 131 ms for only 48 queries, while CAR handle 5 times as many.

The difference in both setups is the number of zombies. With fewer overlaps, the production of zombies cripples CACQ.

In comparison to the static schemes, CAR performs almost as well as TULIP. With 256 queries, the latency of CAR in the greater overlap case is 131 ms as opposed to 113 ms for TULIP. In the fewer overlap case it is 151 ms for CAR as opposed to 147 with TULIP. These results are not surprising as the only difference between CAR and TULIP is cost of adaptivity. Since there are no choices to be made in our experiments, the latency differences we observe lets us reckon the baseline cost of adaptivity.

In summary, our experiments indicate that:

1. The overheads of producing zombies, or unnecessary work, are significant in adaptive dataflows even when relatively fewer zombies are produced.
2. In each scheme, the CAR approach of adaptive precision sharing performs very well.



(a) Fewer overlaps

(b) Greater overlaps

Figure 17: Adaptive query plans: average query latencies

3. In these scenarios, the baseline costs of adaptivity are not very significant.

8 Conclusions

Shared query processing has focused on reducing the overheads of redundancy. Aggressive reduction of repeated work can, however, cause additional wasted work in post-processing useless data.

Thus far, this inherent tension between repeated work and wasted work has been taken for granted. Our major contributions are: (1) to show that this tension is not irreconcilable and (2) To defelop both static and adaptive techniques that balance the tension gracefully.

We defined *precision sharing* as a way to characterize any sharing scheme with neither repeated work, nor wasted work. We then showed how previous work in shared stream processing led to imprecisely shared plans. Armed with these observations we charted a strategy to make static shared plans precise.

Our insight is that *tuple lineage*, an idea from adaptive query processing, is actually more generally applicable. We then proposed TULIP, or "TUple LIneage in static Plans", or technique to make static shared plans precise by using tuple lineage.

Our next contribution was to show how shared adaptive query processors also violate precision sharing. Here we reversed our strategy, and adopted the idea of operator ordering in static dataflows. Our new approach CAR, or "Constrained Adaptive Routing", has almost all the benefits of adaptivity without the side-effects of precision sharing.

Finally we reported a performance study of the various schemes: precise and imprecise, static and adaptive. Our experiments show that the precision sharing approaches either significantly outperform, or are competitive with, all the other schemes under different extreme conditions.

References

[1] R. Avnur *et al.*. Eddies: Continuously adaptive query processing. In *SIGMOD*, pp. 261–272. 2000.

[2] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *CACM*, 13(7):422–426, 1970.

[3] D. Carney, *et al.*. Monitoring streams - a new class of data management applications. In *VLDB*. 2002.

[4] S. Chandrasekaran *et al.*. Streaming queries over streaming data. In *VLDB*. 2002.

[5] S. Chandrasekaran, *et al.*. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*. 2003.

[6] J. Chen, *et al.*. NiagaraCQ: a scalable continuous query system for Internet databases. In *SIGMOD*. 2000.

[7] J. Chen, *et al.*. Design and evaluation of alternative selection placement strategies in optimizing continuous queries. In *ICDE*. 2002.

[8] N. N. Dalvi, *et al.*. Pipelining in multi-query optimization. In *PODS*. 2001.

[9] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object match problem. *Artifical Intelligence*, 19(1):17–37, September 1982.

[10] G. Graefe *et al.*. Dynamic query evaluation plans. In *SIGMOD*, pp. 358–366. 1989.

[11] S. Krishnamurthy, *et al.*. TelegraphCQ: An architectural status report. *IEEE Data Eng. Bull.*, 26(1):11–18, 2003.

[12] S. R. Madden, *et al.*. Continuously adaptive continuous queries over streams. In *SIGMOD*. 2002.

[13] C. Mohan, *et al.*. Single table access using multiple indexes: optimization, execution, and concurrency control techniques. In *EDBT*, pp. 29–43. 1990.

[14] R. Motwani, *et al.*. Query processing, resource management, and approximation in a data stream management system. In *CIDR*. 2003.

[15] V. Raman, *et al.*. Using state modules for adaptive query processing. In *ICDE*. 2003.

[16] T. K. Sellis. Multiple-query optimization. *ACM TODS*, March 1988.

[17] K. Tan *et al.*. Workload scheduling for multiple query processing. *Inf. Proc. Letters*, 55(5):251–257, 1995.

[18] F. Tian *et al.*. Tuple routing strategies for distributed eddies. In *VLDB*, pp. 333–344. 2003.